Netflix's Transition to High-Availability Storage Systems

Siddharth "Sid" Anand

October 2010

1. Abstract

The CAP Theorem states that it is possible to optimize for any 2 of Consistency, Availability, and Network Partition Tolerance, but not all three. Though presented by Eric Brewer in 2000 and proved in 2002, the CAP Theorem has only recently gained significant awareness and that primarily among engineers working on high-traffic applications. With spreading awareness of the CAP theorem, there has been a proliferation of development on AP (a.k.a. Available-Network Partition Tolerant) systems – systems that offer weaker consistency guarantees for higher availability and network partition tolerance. Much of the interest in these AP systems is in the social networking and web entertainment space where eventual consistency issues are relatively easy to mask. Netflix is one company that is embracing AP systems. This paper addresses Netflix's transition to AWS SimpleDB and S3, examples of AP storage systems.

2. Motivation

Circa late 2008, Netflix had a single data center. This single data center raised a few concerns. As a single-point-of-failure (a.k.a. SPOF), it represented a liability – data center outages meant interruptions to service and negative customer impact. Additionally, with growth in both streaming adoption and subscription levels, Netflix would soon outgrow this data center -- we foresaw an imminent need for more power, better cooling, more space, and more hardware.

One option was to build more data centers. Aside from high upfront costs, this endeavor would likely tie up key engineering resources in data center scale out activities, making them unavailable for new product initiatives. Additionally, we recognized the management of multiple data centers to be a complex task. Building out and managing multiple data centers seemed a risky distraction.

Rather than embarking on this path, we chose a more radical one. We decided to leverage one of the leading IAAS (a.k.a. Infrastructure-As-A-Service) offerings at the time, Amazon Web Services (a.k.a. AWS). With multiple, large data centers already in operation and multiple levels of redundancy in various web services (e.g. S3 and SimpleDB), AWS promised better availability and scalability in a relatively short amount of time.

By migrating various network and back-end operations to a 3rd party cloud provider, Netflix chose to focus on its core competency: to deliver movies and TV shows.

3. Migration to the Cloud

Netflix is composed of various applications including but not limited to the following:

- video ratings, reviews, and recommendations
- video streaming
- user registration/log-in
- video queues

Aside from these, there are back-end applications that users do not interact with. These include:

- billing
- DVD disc management (i.e. inventory management and shipping)
- video metadata management (e.g. Movie cast information)

The process of migrating these applications to the cloud is a complicated one and can take months to execute. When planning a migration, one typically chooses a phased strategy with a subset of applications migrating in a particular phase. This limits the scope of changes made at any one time and hence minimizes the risk to the business. Additionally, a phased migration minimizes conflicts with the development of new products that execute in parallel and that might launch first in our old data center.

In order to realize this migration, we needed to answer 2 key questions:

- Which technologies should we adopt in the cloud to persist data?
- While we migrate applications to the cloud, how could we share data between the cloud-based data stores and our DC-resident Oracle instances?

4. Requirements for a Cloud-based Data Store

While surveying available cloud-based data store options, we found ourselves in the middle of a storm of excitement surrounding NoSQL – a movement that loosely encompasses non-relational database storage options. Faced with a large variety of choices, we opted for Amazon Web Services' SimpleDB and S3.

SimpleDB and S3 provide the following:

- Disaster Recovery
- Managed Fail-over and Fail-back
- Distribution (i.e. cross-zone, not cross-region currently)
- Persistence
- Eventual Consistency[1]
- Support for a subset of SQL (i.e. SimpleDB)

Since a key goal of Netflix's cloud migration strategy was to reduce the distraction of scaling and managing complex distributed systems, Amazon's management of SimpleDB and S3 was a key adoption driver. Additionally, SimpleDB's support for an SQL-like language appealed to our application developers as they had mostly worked within the confines of SQL.

After choosing a set of storage systems, we needed to implement a data pipeline to keep these stores synced (i.e. Keeping Oracle synced with S3 and SimpleDB).

[1] SimpleDB's default behavior is to be Eventually Consistent. However, Select and GetAttributes API calls do offer an optional Boolean ConsistentRead flag. Be aware that these calls may be slower and afford lower read availability than Eventually Consistent reads as per the CAP trade-off. As of the writing of this paper, S3 writes in the US Standard (a.k.a. US East) region are eventually consistent. Other regions offer Read-after-write Consistency on Inserts and Eventual Consistency on updates and deletes.

5. Technologies

Before proceeding further, please review this section even if you are familiar with SimpleDB and S3.

5.1. SimpleDB

5.1.1. Data Model

Whereas in RDBMS, data is logically grouped into tables, in SimpleDB, data is grouped into domains. A SimpleDB domain contains zero or more items, just as RDBMS tables contain zero or more rows. Each SimpleDB item has a key field and a value field. Keys are unique within a domain. The value field requires a bit more explanation.

For example, imagine that you have a Soccer Players domain containing 3 items as shown below.

Soccer Players				
Key	**Value**			
ab12ocs12v9	First Name = Harold	Last Name = Kewell	Nickname = Wizard of Oz	Teams = Leeds United, Liverpool, Galatasaray
b24h3b3403b	First Name = Pavel	Last Name = Nedved	Nickname = Czech Cannon	Teams = Lazio, Juventus
cc89c9dc892	First Name = Cristiano	Last Name = Ronaldo		Teams = Sporting, Manchester United, Real Madrid

For item with key = ab12ocs12v9, the value field is a set of attributes. Each attribute has a name and a set of values. For example, this item has an attribute with name First Name and value Harold. Similarly, this item has an Attribute with name Teams and value {Leads United, Liverpool, Galatasaray}. The Teams attribute illustrates that an attribute has a set of values. The curly braces and commas are part of mathematical set notation and convey a set of 3 elements: Leads United, Liverpool, and Galatasaray.

You will notice that the item with key = cc89c9dc892 does not have an attribute with name Nickname. This illustrates another feature of SimpleDB: SimpleDB domains are sparse and hence save space with respect to null values.

Another characteristic of SimpleDB has to do with its consistency model. When a write to SimpleDB is acknowledged, the effect of the write is not immediately visible to all clients. SimpleDB is distributed and will acknowledge a client's write before the write has propagated to all replicas. This is known as eventual consistency. Eventual Consistency is a means to provide good read and write availability by relaxing the consistency guarantee - data will be consistent a short time after a write is acknowledged to the writing client.

SimpleDB does offer an optional Boolean ConsistentRead flag for GetAttributes and Select requests. When this parameter is set to true, applications/users will see the most recently committed version of data. However, as this increased consistency might suffer from lower availability and slower response times, we primarily employ Eventually Consistent reads.

Hence, to summarize SimpleDB's salient characteristics:

- SimpleDB offers a range of consistency options
- SimpleDB domains are sparse and schema-less
- Each item must have a unique Key
- An item contains a set of Attributes
    - Each Attribute has a name
    - Each Attribute has a set of values
- All data is stored as UTF-8 character strings

5.1.2. Operations

SimpleDB operations allow users to either manage domains or manipulate data in domains. Some of these are shown below:

- Domain Management
  - CreateDomain
  - DeleteDomain
  - ListDomains
  - DomainMetadata
- Data Manipulation
  - GetAttributes, Select
  - PutAttributes, BatchPutAttributes
  - DeleteAttributes

5.2. S3

5.2.1. Data Model

A bucket is a container for objects stored in Amazon S3. Every object is contained in a bucket. For example, if the object named photos/skipper.jpg is stored in the gilligans-island bucket, then it is addressable using the URL http://gilligans-island.s3.amazonaws.com/photos/skipper.jpg. As of the writing of this paper, Netflix is deployed in the US Standard region where S3 Object writes are currently Eventually Consistent.

5.2.2. Operations

S3 operations allow users to either manage buckets or manipulate objects. Some of these are shown below:

- Bucket Management
  - GET Bucket
  - PUT Bucket
  - DELETE Bucket
- Object Manipulation
  - GET Object, HEAD Object
  - PUT Object, POST Object
  - DELETE Object

6. Migrating Data To SimpleDB and S3

In the Netflix data center, we primarily use Oracle to persist data. In parts of the movie recommendation infrastructure, we use MySQL. Both are relational databases. In our data center, we do not currently use key-value stores for persistent storage. In order to move data to SimpleDB and S3, we needed to build a data replication system to replicate changes between our data center and AWS, transforming the data model along the way.

Data replication is done via a homegrown framework known as IR (a.k.a. Item Replication). IR's main aim is to keep Oracle data in our data center (a.k.a. DC) in sync with data in Amazon S3 or Amazon SimpleDB. Though this is mainly unidirectional (i.e. data from Oracle is sent to SimpleDB and S3), the use of bidirectional replication is becoming more prevalent.

IR is used at Netflix for the following tasks:

1. Forklifting whole datasets from Oracle tables residing in Netflix's DC to both Amazon SimpleDB

and Amazon S3

2. Replicating incremental data changes as they occur between SimpleDB/S3 and Oracle
    a. This replication is bi-directional in cases where we have multiple masters (i.e. SimpleDB-to-Oracle and Oracle-to-SimpleDB)

An example of Task 1 is the transfer of Netflix Instant Watch Bookmarks to the cloud. Whenever a user stops or pauses a streamed movie, Netflix remembers or "bookmarks" the position in the movie at which he\she stopped. This is called an Instant Watch Bookmark. Netflix remembers these bookmarks so that users may continue movie streaming from where they last left off. All bookmarks for all users were forklifted into SimpleDB.

The Instant Watch Bookmark case also represents a good example of bi-directional incremental replication of data between SimpleDB and Oracle – i.e. Task 2. As part of our phased application migration strategy, we are currently moving device support to the cloud for a few streaming devices (i.e. PS3, Xbox, Wii, etc…) at a time. During the transition period, a user might start watching a movie on a device supported in the cloud and continue watching that same movie on a device supported in our DC. IR keeps these bookmarks in sync.

7. Challenges

The CAP Theorem states that it is possible to optimize for any 2 of Consistency, Availability, and Network Partition Tolerance, but not all three. Though presented by Eric Brewer in 2000 and proved in 2002, the CAP Theorem has only recently gained significant awareness and that primarily among engineers working on high-traffic applications. With spreading awareness of the CAP theorem, there has been a proliferation of development on AP systems – systems that offer weaker consistency guarantees for higher availability and network partition tolerance. Much of the interest in these AP systems is in the social networking and web entertainment space where eventual consistency issues are easy to mask. Systems such as financial systems are not good candidates for such systems.

As attractive as these systems are, the hard truth is that the vast majority in the software industry does not understand how to use them. They understand how to implement systems based on commercially available relational databases. Relational databases handle our consistency problems and offer us convenient features such as the following:

- Transactions
- Locks
- Sequences
- Triggers
- Clocks
- A structured query language (i.e. SQL)
- Database server-side coding constructs (i.e. PL/SQL)
- Constraints

Additionally, most courses and books promote practices such as table normalization -- this results in inter-table relationships and a need to use and understand joins, query optimizers, and foreign key constraints.

Over the past 20 years, the software industry has cultivated a reliance on these features and pushed for the adoption of best practices with the aim of building highly consistent systems. The immense investment towards this one goal (i.e. strong consistency) provides a sizeable barrier to the adoption of AP systems and AP practices.

There are some bright spots in the industry however. A few companies have taken a middle-road – adapting AP practices to a commercial RDBMS. At eBay for example, Oracle was used with the following AP best-practices in place:

- Tables were denormalized to a large extent
- Data sets were stored redundantly, but in different index structures (i.e. several Index-organized Tables)
    - This improved read performance and availability, but required multiple backend writes
    - Though data inconsistencies did arise, they were seldom and were fixed by periodic "fixer" jobs
- DB transactions were forbidden with few exceptions
- Data was sharded among physical instances
- Joins, Group Bys, and Sorts were done in the application layer
- Triggers and PL/SQL were essentially forbidden
- Log-shipping-based (i.e. asynchronous) data replication was done between dedicated Write Masters and a pool of Read Slaves
    - This constituted an eventually-consistent system that traded read-consistency for write-scalability and write-availability
    - For a minority of read-after-write scenarios, developers were told to read from the Write Master
    - In the parlance of distributed computing, consider a case with 12 replicas
        - 1 Write Host and 1 Write Failover – both were written to before acknowledging a write and constituted strongly-consistent replicas
        - 10 Read Hosts – these were written to asynchronously and constituted eventually-consistent replicas
        - To service a read, only one read host was involved, hence R=1
        - To service a write, 2 write hosts were involved, hence W=2
        - Since we are considering 12 replicas in this example, N=12
        - Hence, since R+W<N (i.e. 1+2 < 12), the system is eventually consistent

eBay did a lot to prove the viability of AP practices in practical high-traffic sites. It was middle of the road as developers could still rely on single-host (i.e. not distributed) clocks, sequences, constraints, and locks. Many of the AP systems today force the developer to abandon his/her dependence on these.

Also, in eBay's architecture, a network partition between the Write and Retry host would cause failed writes, thereby limiting write-availability. Many AP systems can handle this using a ring-based consistent hashing scheme to identify a replacement for the unreachable Retry host.

8. SimpleDB vs. S3

Till now, I have discussed moving data from a RDBMS to both S3 and SimpleDB. However S3 and SimpleDB are different data stores that have different strengths.

SimpleDB offers a spin on the typical key-value store model. -- SimpleDB offers data access based on an item key, attribute key pair. Hence, you can address an individual attribute within an item by providing both an item name and attribute name. SimpleDB's API clearly reflects this as the calls are named GetAttributes, PutAttributes, and DeleteAttributes. In a traditional key-value store, one would expect GetItem, PutItem, and DeleteItem API calls.

Additionally, SimpleDB offers a SQL-like language for Selects – this provides more powerful semantics than are available via an item key, attribute key lookup. S3 on the other hand only supports single key lookups for objects. Hence, why would anyone use S3?

One reason has to do with item size limits in SimpleDB. Currently, an item can have no more than 256 attributes and each attribute is limited to 1024 bytes of data. If you need to store more than this amount of data per item key, you can use S3.

Although Netflix transferred some of its RDBMS data to S3, the vast majority was transferred to SimpleDB. The remaining sections will therefore focus on SimpleDB.

9. Best Practices

9.1. Leaving Behind the RDBMS

As mentioned previously, in order to move away from RDBMS, we formulated a set of best practices to deal with the following characteristics of AP systems, specifically SimpleDB:

- Partial or no SQL support. Loosely-speaking, SimpleDB supports a subset of SQL
  - **BEST PRACTICE**
    - Do GROUP BY and JOIN operations in the application layer
    - One way to avoid the need for JOINs is to denormalize multiple Oracle tables into a single logical SimpleDB domain
- No relations between domains
  - **BEST PRACTICE**
    - Compose relations in the application layer
- No transactions
  - **BEST PRACTICE**
    - Use SimpleDB's Optimistic Concurrency Control API: ConditionalPut and ConditionalDelete
- No Triggers
  - **BEST PRACTICE**
    - Do without
- No PL/SQL
  - **BEST PRACTICE**
    - Do without
- No schema - This is non-obvious. A query for a misspelled attribute name will not fail with an error
  - **BEST PRACTICE**
    - Implement a schema validator in a common data access layer
- No sequences
  - **BEST PRACTICE**
    - Sequences are often used as primary keys
      - In this case, use a naturally occurring unique key. For example, in a Customer Contacts domain, use the customer's mobile phone number as the item key
      - If no naturally occurring unique key exists, use a UUID
    - Sequences are also often used for ordering
      - Use a distributed sequence generator
- No clock operations
  - **BEST PRACTICE**
    - Do without
- No constraints. Specifically,
  - No uniqueness constraints
  - No foreign key or referential constraints
  - No integrity constraints
  - **BEST PRACTICE**
    - Application can check constraints on read and repair data as a side effect. This is known as read-repair. Read repair should use the CondtionalPut or ConditionalDelete API to make atomic changes

9.2. New Challenges Encountered in SimpleDB

Additionally, there were some challenges that we encountered specific to SimpleDB. Here are a few:

- SimpleDB domains achieve better write throughput when you shard your datasets across multiple domains
  - **BEST PRACTICE**
    - All datasets at Netflix with significant write load have been partitioned across multiple domains
- No support for native data types. All data is stored and treated as character strings
  - All comparisons (i.e. WHERE clause conditions) and sorts are lexicographical
  - **BEST PRACTICE**
    - Store all date-times in the Joda time (i.e. ISO 8601) format
    - Zero-pad any numeric columns used in sorts or WHERE clause inequality comparisons
- Two separate API calls for DeleteAttributes and PutAttributes
  - Hence, how do you execute an atomic write requiring both a delete of an attribute and an update of another attribute in the same item?
  - **BEST PRACTICE**
    - The most straightforward option is to use pseudo nulls (e.g. a string that reads "NULL").
    - This negates the sparse table optimization of SimpleDB and leads to data bloat
- Getting used to case-sensitivity in domain names and attribute names
  - In many RDBMS, table and column names are case-insensitive. Since SimpleDB does not treat attribute names as case-insensitive, puts, deletes, and selects fail silently. It is up to the programmer to spot case mismatches
  - **BEST PRACTICE**
    - Adopt a convention such as using only upper-case lettering for domain and attribute names
    - Create a data access layer that normalizes all access to attribute and domain names (e.g. TO_UPPER)
- Misspelled attribute names in queries or puts or deletes can fail silently
  - Unlike the case-sensitivity issue, this arises from the lack of domain-specific schema-validation. SimpleDB is not only sparse, it is also schema-free
  - **BEST PRACTICE**
    - Implement a common data access layer. In this layer, implement a schema validator
- Forgetting to specify "limit <N>" in a select query can result in multiple round-trips to get data
  - Multiple round-trips reduce the probability that interactive sites will get all of the data under tight time bounds
  - **BEST PRACTICE**
    - Have the data access layer alter the selects to ensure that "LIMIT <MAX>" is always added
    - The actual value of <MAX> should be easily configurable so that it can be changed if Amazon increases the limit
- Be aware of eventual consistency
  - Avoid the anti-pattern of read-immediately-after-write
  - **BEST PRACTICE**
    - Avoid the read-immediately-after-write anti-pattern
    - If you cannot avoid it, then use ConsistentRead
- Non-indexed queries can be very expensive
  - Anti-pattern : select * from MY_DOMAIN where MY_ATTR_1 is null
  - **BEST PRACTICE**
    - Use a separate flag attribute that is set to TRUE or FALSE based on the nullity of MY_ATTR_1. Hence, the query now uses an index and becomes **select * from MY_DOMAIN where MY_FLAG_ATTR_1 = 'FALSE'**

- Some queries are slow even though indexed
  - Index selectivity matters just like it does in other SQL engines
  - **BEST PRACTICE**
    - As in other database systems, the performance of your select queries is determined by the selectivity of the indexes referenced in your WHERE clause. Make sure that you understand the selectivity of your indexes and that your WHERE clause refers to an attribute whose index is selective
- As in any multi-tenant system, you might experience a significant variation in your response times
  - **BEST PRACTICE**
    - Shield yourself from poor SimpleDB or S3 response times by fronting these persistent stores with caches such as memcached

10. Conclusions

By deciding to leverage SimpleDB and S3, Netflix has been able to rapidly migrate to the cloud while meeting aggressive product launch schedules and supporting high traffic growth. There have been challenges, but we have been able to work around them in most cases. In addition to leveraging SimpleDB and S3, we will continue to explore other data stores. For example, for complex table relationships and complex table access patterns, denormalization is not possible. Hence, a key-value store will not suffice. In such cases, we can use Amazon Relational Database Service (a.k.a. RDS) – Amazon's Hosted MySQL solution. This service would require some amount of DBA-level management however.